1 (1'5 puntos) Indique las características fundamentales de los sistemas tipo UMA, NUMA y cluster. Comente brevemente por qué parece que la tecnología cluster y, más recientemente, la generalización del uso de procesadores gráficos (GPUs) se han acabado imponiendo en el mundo de HPC (computación de alto rendimiento).

SOLUCIÓN

Los dos primeros son máquinas con una visión de memoria compartida. En el caso UMA la memoria está también físicamente compartida y el tiempo de acceso es siempre el mismo. El sistema de intercomunicación suele ser un simples bus, pero tanto éste como la propia memoria se convierten rápidamente en un cuello de botella y permite un número reducido de procesadores (se dice que es poco "escalable"), que en el caso de un bus suele rondar la treintena, aunque con estructuras más sofisticadas aunque poco habituales se ha llegado a pasar de cien.

En el caso NUMA, la visión sigue siendo de memoria compartida, pero está físicamente distribuida entre los procesadores. En este caso, cada procesador tiene un "envoltorio" Hw que consigue la visión de memoria compartida, aunque al final el acceso a direcciones que no están almacenadas localmente se transforme en paso de mensajes por un sistema de intercomunicación heredado de las máquina de memoria distribuida y siempre más sofisticado que un bus, como son los hipercubos, mallas, toros, etc. Estas máquinas disfrutan de la ventaja de la "escalabildiad" de que carecían las UMAs, y llegan a alcanzar más de dos mil procesadores, como es el caso del Cray T3E.

Los clusters son sistemas de computación que unen con una red, normalmente de altas prestaciones, un conjunto de máquinas que son, cada una, de por sí autónomas o que podráin funcionar solas. Estas máquinas pueden ser desde simples PCs de bajo coso hasta, a su vez, MP como los que acabamos de describir, fundamentalmente UMA. Una capa de Sw porporciona una visión de máquina única, con todos los aspectos que aparecen en la llamada SSI, *Single System Image*.

Finalmente, en estos últimos años se ha empezado a utilizar de manera habitual como procesadores matemáticos en las máquinas para HPC la tecnología que se ha venido desarrollando para el procesamiento gráfico, GPUs, y que, al igual que los cluster, brindan la oportunidad de una gran capacidad de cálculo a un precio muy económico, si bien su programación para este propósito sigue resultando un tanto incómoda (CUDA es un buen ejemplo de un Sw diseñado con este fin). En ambos casos, y en la incorporación reciente y de manera "natural" de los procesadores *multicore*, prima sobre todo una cuestión económica o de coste, de modo que, en esencia, y salvo algunas excepciones, no se desarrolla tecnología específica para HPC sino que los sistemas que se emplean en este campo usan tecnología ya existente.

2 (1'5 puntos) Indique en qué consiste la variante de la política de coherencia de cachés mediante invalidación conocida como *Read_Broadcast* y en qué medida puede reducir el número de fallos por invalidación. Explique hasta qué punto sería útil en una situación en que existieran múltiples escritores y lectores accediendo a un mismo bloque. ¿Y en el caso de un solo escritor y varios lectores?

SOLUCIÓN

Si aplica una política de invalidación "pura", el acceso a cada una de las copias de un mismo bloque ha sido modificado por un procesador producirá un fallo (los llamados fallos por invalidación o *invalidation misses*). Con la variante *Read_Broadcast* se reduce a uno el número de este tipo de fallo por cada invalidación que se realice, puesto que se aprovecha la lectura del bloque actualizado por parte del primer procesador que se lo encuentre en estado no válido para actualizar las demás copias.

Esta variante de la política de invalidación resulta ideal en el caso de bloques en los que se acceda por un solo escritor y varios lectores, y se aleja de ese ideal conforme aumenta el número de procesadores que escriben en él, llegando al extremo donde todos los procesadores escriben en un patrón temporal de grano fino a no representar ninguna mejora con respecto a la política de invalidación pura.

3 (2 puntos) Explique el funcionamiento de la instrucción atómica `test&set .Ri, /dir`. Para máquinas de memoria compartida, emplee esta instrucción para implementar la adquisición con espera activa (*spin-lock*) de un cerrojo y de manera que reduzca en lo posible el número de escrituras. Indique cómo se comportaría la implementación propuesta en un sistema de memoria compartida con cachés privadas y en el que se utilizase

invalidación como política de coherencia.

SOLUCIÓN

La instrucción devuelve en Ri el valor previo de la dirección especificada, dir, y escribe un uno en esta posición y esto de manera atómica, i.e., con la garantía de que ningún otro procesador pueda acceder a esta misma dirección mientras dura la ejecución completa del t&s. Este comportamiento lo podemos expresar de la siguiente forma:

```
Test:  test&set  .Ri, /dir
       {
        tmp <-- /dir
        /dir <-- '1'
        Ri <-- tmp
       }
```

supuesto que las acciones comprendidas entre las llaves ocurren atómicamente.

Una primera implementación de una primitiva de adquisición de un cerrojo en la dirección lock podría ser:

```
Test:  test&set  .R1, /lock
       bnz test
       ret
```

que es una implementación con espera activa en la que en cada "vuelta" se fuerza a uno el valor del cerrojo. En la estructura que se plantea, cada procesador tendrá copia del blqoue que contiene la dirección del cerrojo y, consecuentemente, la política de invalidación hará que a cada vuelta al bucle de espera sean invalidadas las copias de los demás procesadores, con los ulteriores fallos de cache por invalidación.

Una posible manera de atenuar esta situación una estructura conocida como test&test&set, y estudiada en las clase de la asignatura, donde primero se consulta (y no se modifica) el valor del cerrojo hasta encontrarlo "abierto" (supuesta una política de coherencia el valor que se lea será el correcto, aunque produzca un fallo por invalidación) y luego tratar de ejecutar el test&set, que si que modificaría el bloque y daría lugar a las invalidaciones pertinentes. Nótese que el número de procesadores que hubiesen pasado el "filtro" de la lectura pudiera ser grande y entonces entonces en el test&set se podría estar en una situación similar a la original, aunque no tiene porque darse este caso normalmente. El pseudocódigo sería como sigue:

```
A:  while (LOAD(lock) = 1) do
nothing;
     if (TEST&SET(lock) = 0)
{
región crítica;
}
     else goto A;
```

Otra posibilidades son los reintentos con retardo o *backoff*, que experimentalmente, y en particular en el caso de un retardo progresivo exponencial, exhiben el mejor rendimiento.

4 (2 puntos) Explique el funcionamiento de la pareja de instrucciones *load_locked / store_conditional* presente en las arquitecturas RISC para construir mecanismos de sincronización en memoria compartida. Implemente la primitiva de adquisición de un cerrojo con espera activa y describa su comportamiento en cuanto al rendimiento supuesto un sistema de memoria compartida, cachés privadas y actualización como política de coherencia de cachés.

SOLUCIÓN

Esta pareja de instrucciones aparece habitualmente en las arquitecturas tipo RISC. Presupone la existencia de cachés privadas y de un mecanismo subyacente de coherencia. La idea es que una dirección a la que se acceda

en lectura con un *load_locked*, ll, se accederá luego en escritura con su instrucción recíproca, *store_conditional*, sc. La unidad de control, UC, no dejará que esta escritura sea efectiva a menos que desde que se realizó la lectura al correspondiente bloque de caché éste no se ha visto modificado o invalidado (entiéndase que sería por el mecanismo de coherencia y la actividad de otro procesador). Este funcionamiento permite implementar de manera sencilla y elegante la primitiva para adquirir un cerrojo según el funcionamiento de test&test&set mencionado en el ejercicio anterior:

```
lock: ll    .R1, /dir-cerrojo  /* ll de la dir dir_cerrojo a R1*/
      bnz   .R1, $lock        /* mientras devuelva 1, sigue leyéndolo*/
      sc   #1, /dir-cerrojo   /* intenta escribir 1 en dir */
      beqz  $lock            /* si no ha tenido éxito, vuelve a leer*/
      ret
```

donde hemos supuesto que el éxito del sc se indica con la activación del flag de cero.

En el caso de que la política de coherencia fuese actualización, la solución propuesta no deja de ser una implementación del citado test&test&set, que, independientemente de qué política se emplee, reduce el número de escrituras y, en consecuencia, el tráfico extra debido al mantenimiento de la coherencia entre las cachés.

5 (3 puntos) Sea un sistema UMA en que se emplea un mecanismo de coherencia de cachés con implementación *snoopy* y protocolo MESI. En este sistema se suceden los tres accesos siguientes y en el mismo orden en que aparecen:

a) Un procesador **i** accede en lectura a su copia local en caché de la variable A cuyo bloque correspondiente se encuentra en estado no válido (I). Suponiendo que dicho bloque está en la caché del procesador **j** en estado exclusivo (S), indique la secuencia de acciones que se da en el sistema para obtener el valor actualizado de A. Señale qué modificaciones se producen en el sistema.

SOLUCIÓN

Se tratará de un **fallo en lectura**, puesto que la copia del bloque aparece en estado I, no válido. Si el bloque está en estado S en j, podría estarlo en otros procesadores también. Supondremos que es j quien suministra la copia válida del bloque. Se darán los siguientes pasos:

1. el procesador i hace una petición a memoria a través del bus
2. la caché j (que ha detectado la petición haciendo *snooping*) pone el valor en el bus (que se arbitra)
3. se abandona el acceso a memoria
4. el procesador i copia en su caché el valor del bus y se pone en estado S
5. todas las otras copias (incluida la de j) permanecen en estado S

a) A continuación el procesador **i** escribe un nuevo valor en A. Indique la secuencia de acciones a que da lugar esta escritura y las modificaciones que conlleva en el sistema.

SOLUCIÓN

Se trata de un **acierto en escritura**, ya que el bloque se encuentra en estado S. Se darán los siguientes pasos:

1. el procesador i difunde en el bus una operación de invalidación
2. los procesadores (haciendo *snooping*) con copia en estado S (entre los que se encuentra seguro j, supuesto que el bloque no haya sido reemplazado localmente) transitan: S->I (no válido)
3. se actualiza el valor local en la caché
4. cambio en el estado local en i: S->M (modificado, única copia válida)

a) En un instante posterior, el mismo procesador **j** accede en lectura a variable A, cuyo bloque correspondiente *no* ha sido reemplazado de su caché. Indique la nueva secuencia de acciones a que da lugar esta lectura y las modificaciones a que da lugar.

SOLUCIÓN

Se trata de un nuevo **fallo en lectura**, ahora en el procesador j y con la única copia válida (estado M) en i. Se darán los siguientes pasos:

1. el procesador j hace una petición a memoria a través del bus
2. la caché i, haciendo *snooping*, pone el valor en el bus
3. se abandona el acceso a memoria
4. el procesador local j copia el valor en su caché
5. la copia local en j se pone en estado S
6. el valor origen (M) se actualiza en memoria: *copy-back*
7. el bloque origen en i cambia de estado: M -> S

1 (2 puntos) OpenMP permite especificar distintos esquemas de planificación con la clausula `schedule`. Las opciones admitidas son `static`, `dynamic` y `guided`, ¿cuál se espera en general que tenga mejor rendimiento y por qué? En relación con esta cláusula ¿a qué se denomina *chunk* y para qué se utiliza?

SOLUCIÓN

La opción de planificación que debe comportarse mejor en un rango mayor de situaciones es `guided`, ya que intenta aprovechar las ventajas de las otras dos. A la hora de planificar las iteraciones de un bucle, comienza seleccionando bloques grandes de iteraciones, de modo que evita estar constantemente planificando (como podría ocurrir al emplear la opción `dynamic`). Por otro lado, `guided` evita los desequilibrios de carga presentes en `static`, ya que según se va avanzando en el bucle los bloques de iteraciones son más pequeños, con lo que se evita que haya procesadores ociosos mientras otros mantienen carga de trabajo.

Se denomina *chunk* a un parámetro opcional de la cláusula `schedule` que establece el número mínimo de iteraciones que se asignará a cada bloque (de iteraciones) a la hora de planificar.

2 (2 puntos) Explique el funcionamiento de la instrucción atómica `lock.xchg .Ri, /dir` e implemente para máquinas de memoria compartida un cerrojo con espera activa (*spin-lock*) empleando dicha instrucción.

SOLUCIÓN

Esta instrucción (*exchange* con el prefijo lock) está presente en algunas arquitecturas para que se puedan construir mecanismos de sincronización de memoria compartida de mayor abstracción, como pueden ser por ejemplo los cerrojos (*locks*) y las barreras (*barriers*).

Su ejecución con los operandos propuestos intercambia el contenido del registro Ri por el de la posición de memoria /dir de manera *atómica* o sin que ningún otro procesador pueda acceder a esa dirección hasta que termine de ejecutarse.

Una posible implementación de una primitiva para ganar, con la llamada espera activa, el acceso a un cerrojo podría ser, simplemente, tratar de reproducir el comportamiento de una instrucción de `test&set`, tal y como se desarrolló en una de las sesiones de ejercicios de la asignatura. Por simplicidad, podemos suponer que la ejecución de la instrucción afecta a los flags aritméticos (según el valor que retorna el registro Ri, en nuestro caso mientras no devuelva el valor cero) y, también, que se trata de una arquitectura genérica que permita los direccionamientos que empleamos en el código.

```
          LD .R0, #D'01
intento:  lock.xchg .R0, /dir_cerrojo
          bnz $intento
          ret
```

donde se debe notar que no haría falta volver a llevar un uno al registro empleado en cada vuelta (o *spin*) (puesto que devuelve ese valor en el registro R0) y que una implementación más eficiente sería la que hiciese primero la comprobación en lectura de que el cerrojo está abierto y luego intentase realizar su cierre en acceso atómico, lo que denominamos *test-and-test-and-set*.

3 (3 puntos) Para la implementación de una primitiva de LOCK(cerrojo) en un MP UMA con cachés privadas y un mecanismo de coherencia de cachés por *actualización* se consideran las tres opciones siguientes, A, B y C:

```
Implementación A

lock:
     while (LOAD(cerrojo) == 1) do
                nothing;
     STORE (1, cerrojo); almacena un '1' en la dirección del cerrojo

Implementación B

lock:
     while (TEST&SET (cerrojo) == 1) do
                nothing;

Implementación C

lock:
     repeat
         while (LOAD(cerrojo) == 1) do
                nothing;
     until (TEST&SET(cerrojo) == 0)
```

a) Explique brevemente el funcionamiento de las tres implementaciones. ¿Son todas implementaciones correctas de la primitiva LOCK para el sistema considerado? Indique razonadamente por qué.

b) Para cada una de las implementaciones que considere que funcionan correctamente, explique en qué medida es eficiente desde un punto de vista del rendimiento y por qué.

SOLUCIÓN

a) Si se analizan en un primer vistazo las tres soluciones, se puede observar que en todos los casos tienen una estructura de *spin-lock*. En **A** se consulta con `load` el valor del cerrojo hasta que sea 0 y entonces se escribe 1 en la dirección correspondiente con `store`. El mecanismo de coherencia haría que esta escritura se hiciese visible en el resto de las cachés que tuviesen copia del bloque del cerrojo. Sin embargo, desde la lectura hasta la escritura no existe la garantía de que se realice sin que pueda mediar la escritura de otro procesador y, por tanto, se trata de una implementación incorrecta. En **B** el `test&set` devuelve el valor previo de la dirección y escribe indiscriminadamente un uno en ella, y con la garantía de que estas dos acciones se realizan de manera atómica, a diferencia del caso anterior y, en consecuencia, sí se trata de una realización correcta. Por último, en **C** se realiza primero la comprobación de que el cerrojo está abierto y sólo cuando así sea se ejecuta un `test&set` para tratar de ganar su acceso, en un comportamiento que es entonces análogo al caso **B** (aunque sólo para los procesadores que hubieran pasado la comprobación primera) y por tanto funcionalmente correcto.

b) Sólo consideraremos aquí las implementaciones correctas, o sea, B y C. Con *actualización* como política de coherencia de cachés todas las copias de un mismo bloque se mantienen siempre actualizadas. En el caso de **B**, todos los procesadores detenidos en el lock darán lugar a que se produzca tráfico de actualización en el sistema debido a las escrituras que se propagan. Nótese, sin embargo, que no se producirían fallos de caché adicionales a los que ocurrirían en un comportamiento "monoprocesador": si un bloque se encuentra en caché será siempre copia válida. En **C** los procesadores en el lock no producirán fallos de caché en la primera fase (*spin* en la lectura del cerrojo) y, por tanto, ahora se reduce significativamente el tráfico extra debido a la ausencia de actualizaciones.

También es interesante destacar aquí que el patrón de *sharing* o "compartición" que se presupone en el acceso a un cerrojo es habitualmente de "grano fino", independientemente de que se haya reducido el número de escrituras. En el caso de una potencial "compartición secuencial", la actualización no parece la política de coherencia adecuada, por el exceso de tráfico que añade innecesariamente en el sistema de interconexión.

4 (1 punto) Indique la repercusión de los procesadores *multicore* y de las tarjetas gráficas (GPUs) en el mundo de la computación de altas prestaciones (HPC).

SOLUCIÓN

Estas dos tecnologías están teniendo un impacto fundamental en el campo del HPC. Hoy en día la práctica totalidad de los sistemas en el ránking Top500 tienen una estructura *multicore* (en definitiva, un multiprocesador en un chip), y, en su momento, hace un par de años, fue una innovación que contribuyó fundamentalmente a batir la barrera de los PFLOPS en computación de alto rendimiento.

El uso de los procesadores gráficos (GPU) como hardware específico para realizar operaciones masivas en coma flotante ha representado también un cambio cuantitativo fundamental en HPC, tanto es así que tres de las cinco primeras máquinas en el Top500 (el llamado "Top5"), en su última edición, emplean este tipo de procesadores. Aunque presentan cierta dificultad de programación, librerías como CUDA permiten que sin demasiado esfuerzo se le saque partido en computación científica a este tipo de procesadores. Su virtud fundamental es obtener rendimientos del orden TFLOPS a un coste muy reducido, algo impensable hasta hace muy poco.

5 (2 punto) Describa los estados en que puede estar un bloque de memoria caché en el protocolo de coherencia de cachés basado en invalidación llamado MESI.

SOLUCIÓN

Podríamos resumir los estados del modo siguiente. Hay dos estados, digamos, "extremos", el estado I (Invalid), en que el bloque no es válido y el estado M (Modified), en que es la única copia válida en toda la máquina (y que en su momento dará pie a que se actualice en memoria principal y/o a servir el fallo por invalidación que se dé en otro procesador). Los otros dos son el estado E (Exclusive), que representa una especie de paso previo al estado M (el bloque es también la única copia en caché, pero es coherente con la copia de memoria, que por lo tanto no necesitaría ser actualizada en *copy-back*), y el S (Shared), en que el bloque está potencialmente copiado en otras cachés y también es coherente con la copia de memoria. En este estado las escrituras se deberán difundir por el bus para que esas otras copias se invaliden.